

Manipulating Java Programs

Dirk Bäumer

IBM Research, Switzerland
dirk_baeumer@ch.ibm.com

Philippe Mulet

IBM Parislab, France
philippe_mulet@fr.ibm.com

Outline

- A guided tour through code manipulation services
 - Java Model
 - Search Engine
 - **Abstract Syntax Tree & AST Rewriting**
 - Refactoring
- Each part is associated with examples
 - *Encapsulate field*: create getter/setter methods for a field and use only those to access the field

```
class Example {  
    public String name;  
  
    public void main(String[] args) {  
        System.out.println(example.name);  
    }  
}
```

```
class Example {  
    private String name;  
  
    public String getName() { return name; }  
    public void setName(String name) { ... }  
  
    public void main(String[] args) {  
        System.out.println(example.getName());  
    }  
}
```

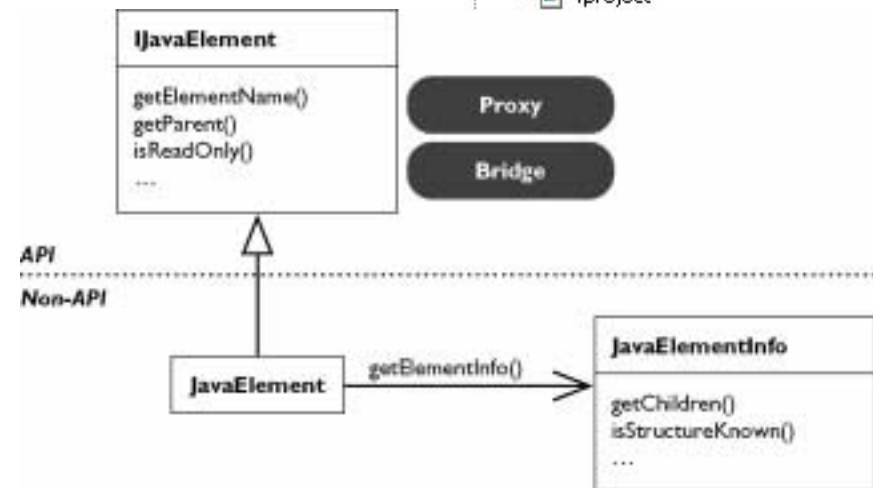
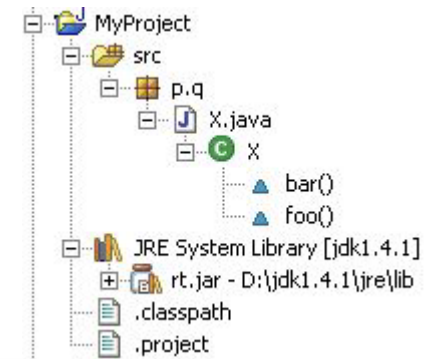
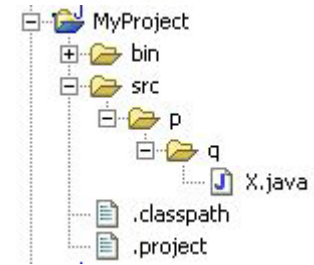
- *Rename method*: rename the method's declaration and update all references to it

The Java Model – org.eclipse.jdt.core

- Render entire workspace from Java angle
 - using .classpath
 - classpath entry is package fragment root
 - can even denote JAR outside workspace
 - granularity down to individual fields or methods

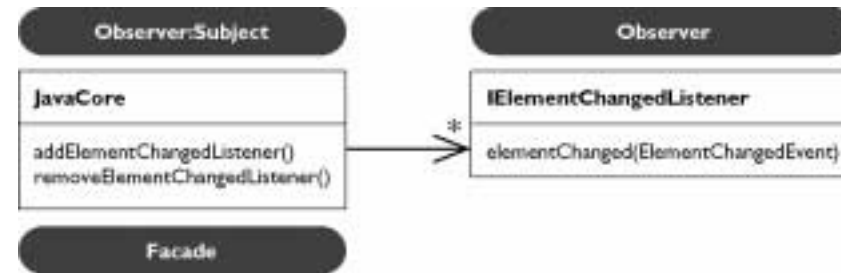
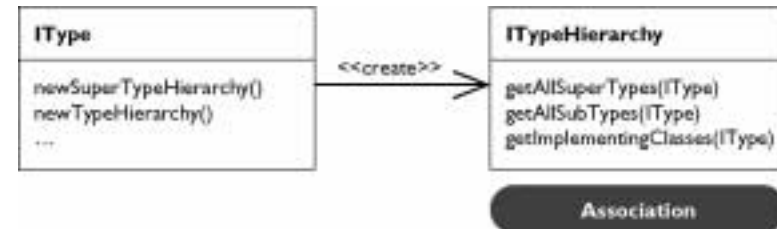
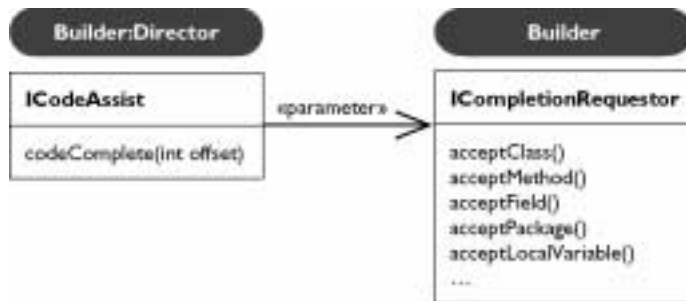
- Pure source model
 - accurate independently of build actions
 - fault-tolerant
 - no resolved information

- Handle/Info design
 - scalability: model non exhaustive
 - info lazily populated, LRU cache
 - stable handle



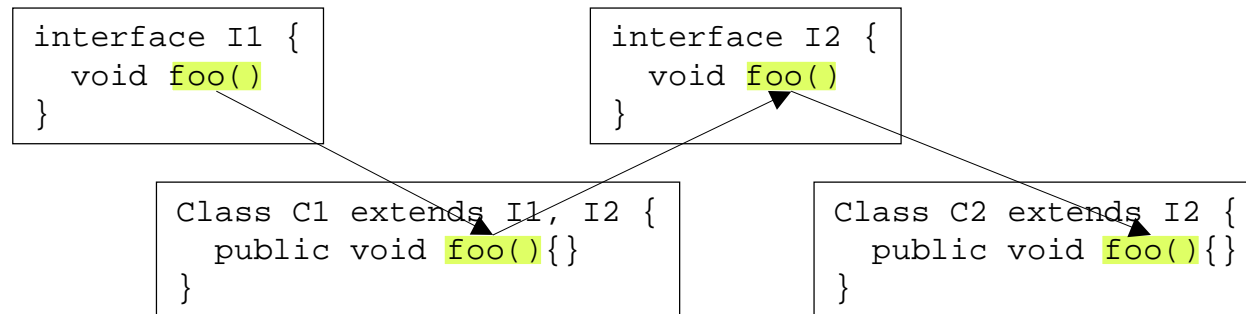
The Java Model Offering

- Navigation
- Type hierarchies
- Self-updating with change notification
- Basic modifying operations
- Many more...
 - code completion, selection
 - formatting, sorting
 - resolving, evaluating



Using the Java Model

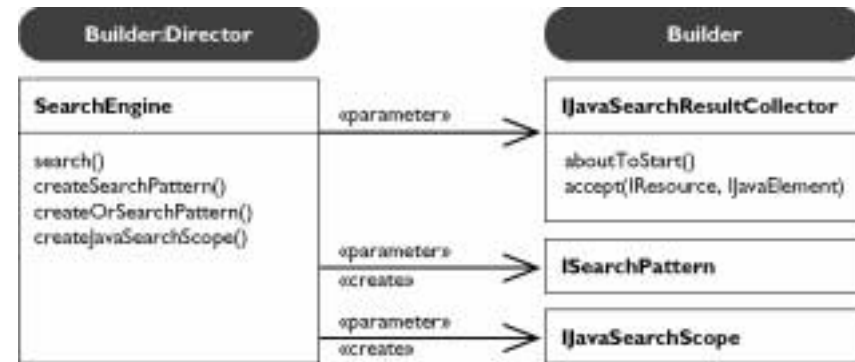
- Check status of Java element to be manipulated
 - is the element declared in a class file, is it final, ...
 - is the code syntactically correct
- Type hierarchies – rename method
 - ripple effect computation



- check if renamed method overrides existing method or is overridden in subclass

The Search Engine – org.eclipse.jdt.core.search

- Pattern-based
 - construct: package, type, field, method, constructor
 - occurrence: declaration, reference, read/write-access, extends/implements
 - wildcards: find all references to any method foo (any arguments)
 - pattern can also be based on Java element
- Scoped search
 - scope = set of Java elements
 - predefined workspace and hierarchy scopes
- Match contains details
 - range information, accuracy, enclosed element
- Indexing
 - pure source, updated automatically in background
 - built-in support to perform concurrent queries
 - looking up in indexes to find possible document matches
- Locating
 - document matches are processed to issue actual search matches



Using the Search Engine

- Find references to affected code
- Anticipate semantic changes – rename method

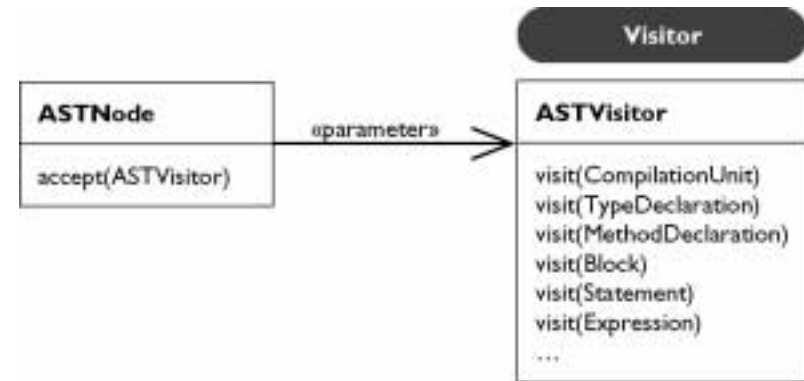
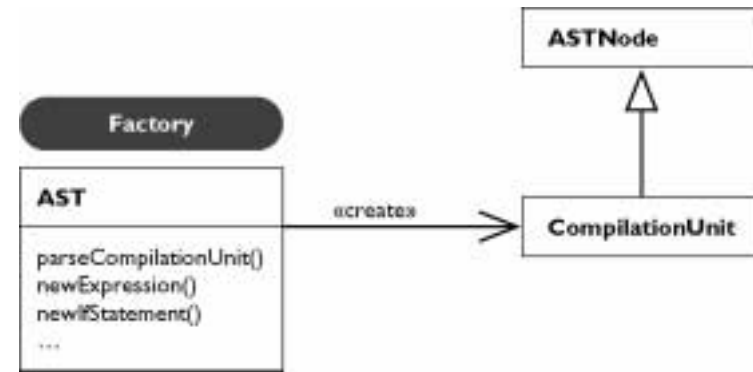
```
public class Example {
    public void set(Object o) { ... };
    public void setString(String...s);{ ... };    Rename setString to set

    void client() {
        set(new String());
    }
}
```

- Compute set of conflicting elements: {Example#set(Object)}
 - name of the conflicting element is equal to new name
 - conflicting element is overloaded, shadowed or hidden by the refactored element
- Determine references to conflicting elements: set1= {Example#client()}
- Perform the refactoring in memory
- Determine references to conflicting elements: set2= { }
- A semantic change occurred if set1 != set2

Abstract Syntax Tree – org.eclipse.jdt.core.dom

- Zooming in a compilation unit
 - a node for every construct in the language
 - source range
 - editable, with some simple validation
- Pure source model
 - instant life-cycle, fully populated
 - can resolve all references to bindings
 - bindings are fully connected
- Analysis using visitor
- Limitations in 2.1
 - cannot persist edits
 - cannot reify all comments
 - cannot create AST on arbitrary sources



Using the Abstract Syntax Tree

- Perform advanced precondition checks – encapsulate field

Original Code

```
foo(field++);

public void visit(PostfixExpression node) {
    checkIfNodeRepresentsField();
    ASTNode parent= node.getParent();
    if (!(parent instanceof ExpressionStatement))
        warnUser();
    ...
}
```

Refactored Code

```
int temp= getField();
setField(temp + 1);
foo(temp);
```

- Gather additional information to manipulate code – encapsulate field

```
foo(++field);

public void visit(PrefixExpression node) {
    checkIfNodeRepresentsField();
    ASTNode parent= node.getParent();
    if (!(parent instanceof ExpressionStatement))
        fSetterMustReturnValue= true;
    ...
}
```

AST Rewriting – org.eclipse.jdt.core.dom

- Instead of manipulating the source code change the AST and write changes back to source
- Descriptive approach
 - describe changes without actually modifying the AST
 - allow reuse of the AST over several operations
 - support generation of a preview
- Modifying approach
 - directly manipulates the AST
 - API is more intuitive
 - implemented using the descriptive rewriter
- Rewriter characteristics
 - preserve user formatting and markers
 - generate an edit script

Using AST Rewriting

Original Code

```
public class Example {
    String field;
    public void foo() {
        field= "EclipseCon" + /*...*/
            "2004";
    }
}
```

Refactored Code

```
public class Example {
    String field;
    public void foo() {
        setField("EclipseCon" + /*...*/
            "2004");
    }
}
```

Code snippet demonstrating the rewriter

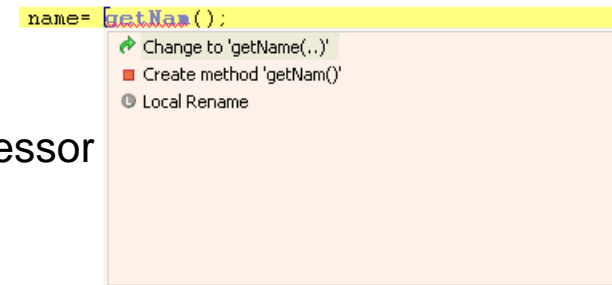
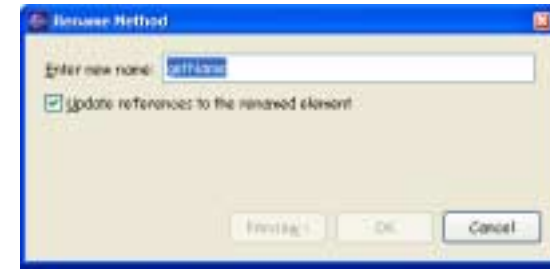
```
final ASTRewrite rewrite= new ASTRewrite(root);
root.accept(new ASTVisitor() {
    public boolean visit(Assignment assignment) {
        // check if affected
        AST ast= assignment.getAST();
        MethodInvocation setter= ast.newMethodInvocation();
        setter.setName(ast.newSimpleName(setterName));
        setter.arguments().add(
            rewrite.createMoveTarget(assignment.getRightHandSide()));
        rewrite.replace(assignment, setter);
    }
});

TextEdit edit= rewrite.rewriteAST(document);
```

Code Manipulation Toolkits

- Refactoring – org.eclipse.refactoring
 - refactorings - org.eclipse.refactoring.core.Refactoring
 - responsible for precondition checking
 - create code changes
 - code changes - org.eclipse.refactoring.core.Change
 - provide Undo/Redo support
 - support non-textual changes (e.g. renaming a file)
 - support textual changes based on text edit support
 - user interface is dialog based

- Quick fix & Quick Assist – org.eclipse.jdt.ui.text.java
 - AST based
 - processors - org.eclipse.jdt.ui.text.java.IQuickFixProcessor
 - check availability based on problem identifier
 - generate a list of fixes
 - user interface is provided by editor



Open-up Search – `org.eclipse.jdt.core.search`

- Evolve search into a framework for searching in Java-like languages:
 - JSP, SQLJ, ...
- Framework provides:
 - pluggable search participant, e.g. *JavaParticipant*
 - built-in index infrastructure
 - scheduling queries and background indexing activity
 - extensible patterns
 - combining multiple participants in one search
- Each participant describes specifics for a language
 - indexing: parsing a document for indexing
 - locating: resolve and find matches in one document
- Participants can cooperate and delegate to each other
 - e.g., translate JSP to Java equivalent, and delegate to Java participant

Open-up Refactoring – `org.eclipse.refactoring.core`

- Provide language independent refactoring API
- Enable other plug-in to participate in certain refactorings
 - generic refactorings for rename, move, delete, ...
 - update JSPs when renaming types, methods, ...
 - update `` tags when moving HTML file
 - language specific refactorings
 - update JSPs and SQLJ when changing method signature
- Refactoring is split into
 - one processor
 - comparable to a single “closed” refactoring
 - define user interface
 - 0..n participants
 - each participant provides domain specific precondition checking and code changes
 - cannot contribute to the user interface

Summary

- JDT delivers powerful program manipulation services
 - Java Model, Search engine and DOM AST
 - provided through Eclipse Java IDE
 - but also in headless mode (can be used programmatically)
 - Visual Editor, EMF, metric tools, ...
- New in 3.0
 - AST rewriting API
 - Refactoring & Quick Fix becomes API
 - ability to participate in Search & Refactoring
- Community feedback is essential
 - bug reports: <http://bugs.eclipse.org/bugs>
 - mailing lists: <http://www.eclipse.org/mail/index.html>
 - newsgroups: <news://news.eclipse.org/eclipse.tools.jdt>

Questions

Martin Aeschlimann

Jérôme Lanneluc

Kai-Uwe Mätzel

André Weinand

Kent Johnson

Tom Eicher

Frédéric Fusier

Jim des Rivières

Philippe Mulet

David Audel

Olivier Thomann

Adam Kiezun

Christof Marti

Daniel Megert

Markus Keller

Erich Gamma

Dirk Bäumer